

15XT98 Data Mining Lab

Final Package Report

Name: **Subhashini H (16PT36)** and **Sivanesan D (16PT35)**

Research Paper: **Collaborative Filtering for Implicit Feedback Datasets**

Date: 15.11.2020

Contents

1. Summary
2. Introduction
3. Alternating Least Squares
4. Processing the Data
5. Creating a Training and Validation Set
6. Implementing ALS for Implicit Feedback
7. Evaluating the Recommender System
8. A Recommendation Example (Check)
9. Conclusion
10. References

1. Summary

Nowadays, people used to buy products online more than from stores. Previously, people used to buy products based on the reviews given by relatives or friends but now as the options increased and we can buy anything digitally we need to assure people that the product is good and they will like it. To give confidence in buying the products, recommender systems were built. This package focuses on

building recommender system for **implicit feedback datasets using alternating least square method** from the research paper mentioned above.

2. Introduction

Recommendation system is a personalized information filtering technology used to either predict whether a particular user will like a particular item or to identify a set of N items that will be of interest to a certain user. There are various techniques and approaches used by the recommender system: User-based approach, Item based approach and Hybrid recommendation approaches. The recommender system apply data mining techniques and prediction algorithms to predict user's interest on information, product and services.

Typically, a recommender system compares a user profile to some reference characteristics, and seeks to predict the 'rating' or 'preference' that a user would give to an item they had not yet considered. These characteristics may be from the information item (the content-based approach) or the user's social environment (the collaborative filtering).

2.1 Collaborative Filtering

This is based on the relationship between users and items, with no information about the users or the items required! All we need is a rating of some kind for

each user/item interaction that occurred where available. There are two kinds of data available for this type of interaction: explicit and implicit.

- Explicit: A score, such as a rating or a like
- Implicit: Not as obvious in terms of preference, such as a click, view, or purchase

A common task of recommender systems is to improve customer experience through personalized recommendations based on prior implicit feedback. These systems passively track different sorts of user behavior, such as purchase history, watching habits and browsing activity, in order to model user preferences. Unlike the much more extensively researched explicit feedback, we do not have any direct input from the users regarding their preferences. In particular, we lack substantial evidence on which products consumer dislike.

Since more data usually means a better model, implicit feedback is where our efforts should be focused. While there are a variety of ways to tackle collaborative filtering with implicit feedback, we will focus on the method included in Spark's library used for collaborative filtering, **alternating least squares (ALS)**.

3. Alternating Least Squares

To figure out how the users and the items are related to each other we can apply **matrix factorization**. We take a large matrix of user/item interactions and figure out the latent features that relate them to each other in a much smaller matrix of user features and item features. That's exactly what ALS is trying to do through matrix factorization.

As the image below demonstrates, let's assume we have an original ratings matrix R of size $M \times N$, where M is the number of users and N is the number of items. This matrix is quite sparse, since most users only interact with a few items each. We can factorize this matrix into two separate smaller matrices: one with dimensions $M \times K$ which will be our latent user feature vectors for each user (U) and a second with dimensions $K \times N$, which will have our latent item feature vectors for each item (V). Multiplying these two feature matrices together approximates the original matrix, but now we have two matrices that are dense including a number of latent features K for each of our items and users.

	Item 1	Item 2	Item 3	Item 4	Item 5	Item 6
User 1	X		X		X	
User 2		X	X			
User 3				X		X
User 4					X	
User 5	X	X		X		X
User 6			X	X		
User 7	X	X	X		X	X
User 8		X		X		
User 9			X			

R

\approx

	UF1	UF2
User 1		
User 2		
User 3		
User 4		
User 5		
User 6		
User 7		
User 8		
User 9		

U

X

V

	Item 1	Item 2	Item 3	Item 4	Item 5	Item 6
IF1						
IF2						

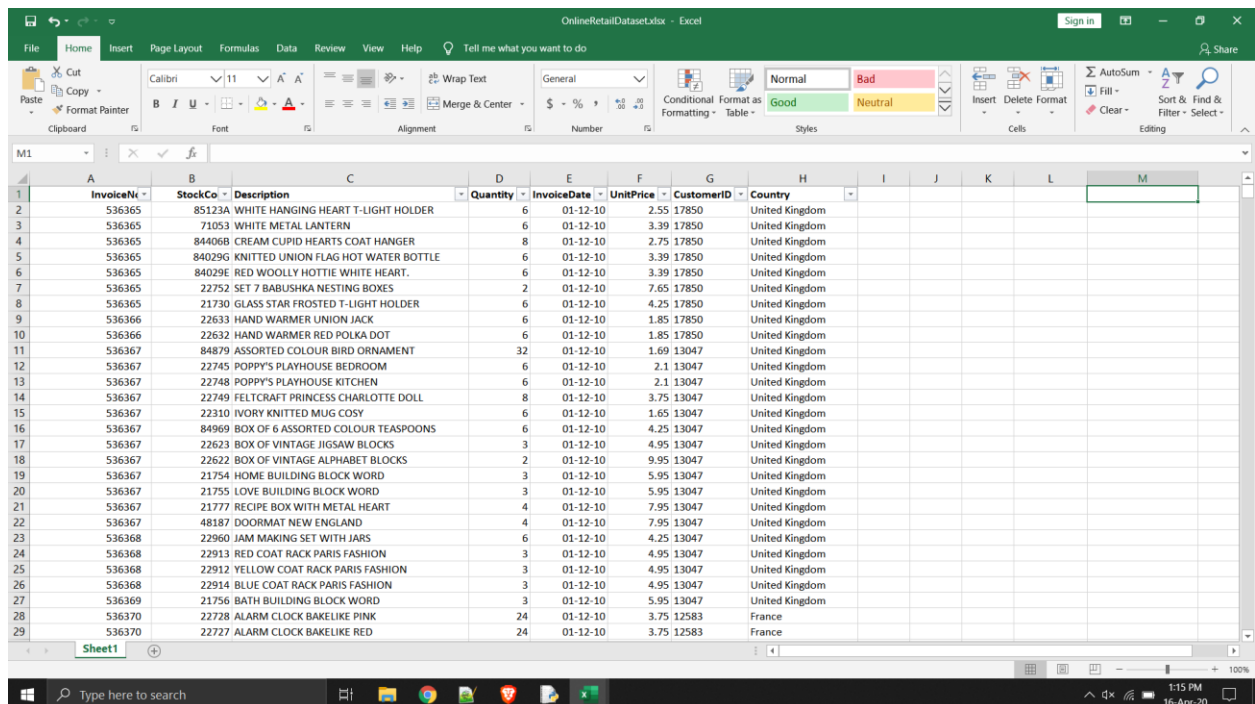
In order to solve for U and V , we could either utilize **SVD** to solve the factorization more precisely or apply ALS to approximate it. In the case of ALS, we only need to solve one feature vector at a time, which means it can be run in parallel! (large advantage). To do this, we can randomly initialize U and solve for V . Then we can go back and solve for U using our solution for V . We keep iterating back and forth like this until we get a convergence that approximates R as best as we can.

After this has been finished, we can simply take the dot product of U and V to see what the predicted rating would be for a specific user/item interaction, even if there was no prior interaction. This basic methodology was adopted for implicit feedback problems in the paper [Collaborative Filtering for Implicit Feedback](#)

[Datasets](#) by Hu, Koren, and Volinsky. We will use this paper’s method on a real dataset and build recommender system.

4. Processing the Data

The data used is from the **UCI Machine Learning repository**. The dataset is called “**Online Retail**”. It is the sales data of the customers for 2 years. The data has around 4 lakhs transactions. The data has approximately 4500 customers and 3000 items. Below is the snapshot of the dataset we used. Here we mainly consider the **Customer Id** and **Stock Code (Product Id)** from the dataset for our work.



	A	B	C	D	E	F	G	H	I	J	K	L	M
	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country					
1													
2	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	01-12-10	2.55	17850	United Kingdom					
3	536365	71053	WHITE METAL LANTERN	6	01-12-10	3.39	17850	United Kingdom					
4	536365	844068	CREAM CUPID HEARTS COAT HANGER	8	01-12-10	2.75	17850	United Kingdom					
5	536365	840296	KNITTED UNION FLAG HOT WATER BOTTLE	6	01-12-10	3.39	17850	United Kingdom					
6	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	01-12-10	3.39	17850	United Kingdom					
7	536365	22752	SET 7 BABUSHKA NESTING BOXES	2	01-12-10	7.65	17850	United Kingdom					
8	536365	21730	GLASS STAR FROSTED T-LIGHT HOLDER	6	01-12-10	4.25	17850	United Kingdom					
9	536366	22633	HAND WARMER UNION JACK	6	01-12-10	1.85	17850	United Kingdom					
10	536366	22632	HAND WARMER RED POLKA DOT	6	01-12-10	1.85	17850	United Kingdom					
11	536367	84879	ASSORTED COLOUR BIRD ORNAMENT	32	01-12-10	1.69	13047	United Kingdom					
12	536367	22745	POPPY'S PLAYHOUSE BEDROOM	6	01-12-10	2.1	13047	United Kingdom					
13	536367	22748	POPPY'S PLAYHOUSE KITCHEN	6	01-12-10	2.1	13047	United Kingdom					
14	536367	22749	FELTCRAFT PRINCESS CHARLOTTE DOLL	8	01-12-10	3.75	13047	United Kingdom					
15	536367	22310	IVORY KNITTED MUG COSY	6	01-12-10	1.65	13047	United Kingdom					
16	536367	84969	BOX OF 6 ASSORTED COLOUR TEASPOONS	6	01-12-10	4.25	13047	United Kingdom					
17	536367	22623	BOX OF VINTAGE JIGSAW BLOCKS	3	01-12-10	4.95	13047	United Kingdom					
18	536367	22622	BOX OF VINTAGE ALPHABET BLOCKS	2	01-12-10	9.95	13047	United Kingdom					
19	536367	21754	HOME BUILDING BLOCK WORD	3	01-12-10	5.95	13047	United Kingdom					
20	536367	21755	LOVE BUILDING BLOCK WORD	3	01-12-10	5.95	13047	United Kingdom					
21	536367	21777	RECIPE BOX WITH METAL HEART	4	01-12-10	7.95	13047	United Kingdom					
22	536367	48187	DOORMAT NEW ENGLAND	4	01-12-10	7.95	13047	United Kingdom					
23	536368	22960	JAM MAKING SET WITH JARS	6	01-12-10	4.25	13047	United Kingdom					
24	536368	22913	RED COAT RACK PARIS FASHION	3	01-12-10	4.95	13047	United Kingdom					
25	536368	22912	YELLOW COAT RACK PARIS FASHION	3	01-12-10	4.95	13047	United Kingdom					
26	536368	22914	BLUE COAT RACK PARIS FASHION	3	01-12-10	4.95	13047	United Kingdom					
27	536369	21756	BATH BUILDING BLOCK WORD	3	01-12-10	5.95	13047	United Kingdom					
28	536370	22728	ALARM CLOCK BAKELIKE PINK	24	01-12-10	3.75	12583	France					
29	536370	22727	ALARM CLOCK BAKELIKE RED	24	01-12-10	3.75	12583	France					

We need to take all of the transactions for each customer and put these into a format ALS can use. This means we need each unique customer ID in the rows of the matrix, and each unique item ID in the columns of the matrix. The values of the matrix should be the total number of purchases for each item by each customer.

The dataset includes the invoice number for different purchases, along with the StockCode (or item ID), an item description, the number purchased, the date of purchase, the price of the items, a customer ID, and the country of origin for the customer.

```
import pandas as pd
import scipy.sparse as sparse
import numpy as np
from scipy.sparse.linalg import spsolve
```

```
!pip install implicit
```

```
website_url = 'http://archive.ics.uci.edu/ml/machine-learning-databases/00352/Online%20Retail.xlsx'
retail_data = pd.read_excel(website_url) # This may take a couple minutes
```

```
retail_data.head()
```

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
0	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	2010-12-01 08:26:00	2.55	17850.0	United Kingdom
1	536365	71053	WHITE METAL LANTERN	6	2010-12-01 08:26:00	3.39	17850.0	United Kingdom
2	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	2010-12-01 08:26:00	2.75	17850.0	United Kingdom
3	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	2010-12-01 08:26:00	3.39	17850.0	United Kingdom
4	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	2010-12-01 08:26:00	3.39	17850.0	United Kingdom

In the below code snippet, we are checking if there are any missing values in the data.

```
retail_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 541909 entries, 0 to 541908
Data columns (total 8 columns):
InvoiceNo      541909 non-null object
StockCode      541909 non-null object
Description     540455 non-null object
Quantity       541909 non-null int64
InvoiceDate    541909 non-null datetime64[ns]
UnitPrice      541909 non-null float64
CustomerID     406829 non-null float64
Country        541909 non-null object
dtypes: datetime64[ns](1), float64(2), int64(1), object(4)
memory usage: 33.1+ MB
```

Most columns have no missing values, but **Customer ID is missing in several rows**. If the customer ID is missing, we don't know who bought the item. We should drop these rows from our data first. We can use the `pd.isnull()` to test for rows with missing data and only keep the rows that have a customer ID.

```
cleaned_retail = retail_data.loc[pd.isnull(retail_data.CustomerID) == False]
```

```
item_lookup = cleaned_retail[['StockCode', 'Description']].drop_duplicates() # Only get unique item/description pairs
item_lookup['StockCode'] = item_lookup.StockCode.astype(str) # Encode as strings for future lookup ease
```

```
item_lookup.head()
```

	StockCode	Description
0	85123A	WHITE HANGING HEART T-LIGHT HOLDER
1	71053	WHITE METAL LANTERN
2	84406B	CREAM CUPID HEARTS COAT HANGER
3	84029G	KNITTED UNION FLAG HOT WATER BOTTLE

We create a lookup table that keeps track of each item ID along with a description of that item. This can tell us what each item is, such as that Stock Code 71053 is a white metal lantern. Now that this has been created, we need to:

- Group purchase quantities together by stock code and item ID
- Change any sums that equal zero to one (this can happen if items were returned, but we want to indicate that the user actually purchased the item instead of assuming no interaction between the user and the item ever took place)
- Only include customers with a positive purchase total to eliminate possible errors
- Set up our sparse ratings matrix

Our matrix is going to contain thousands of items and thousands of users with a user/item value required for every possible combination. That is a LARGE matrix, so we can save a lot of memory by keeping the matrix sparse and only saving the locations and values of items that are not zero to reduce the memory. The code below will finish the preprocessing steps necessary for our final ratings sparse matrix:

```

cleaned_retail['CustomerID'] = cleaned_retail.CustomerID.astype(int) # Convert to int for customer ID
cleaned_retail = cleaned_retail[['StockCode', 'Quantity', 'CustomerID']] # Get rid of unnecessary info
grouped_cleaned = cleaned_retail.groupby(['CustomerID', 'StockCode']).sum().reset_index() # Group together
grouped_cleaned.Quantity.loc[grouped_cleaned.Quantity == 0] = 1 # Replace a sum of zero purchases with a one to
# indicate purchased
grouped_purchased = grouped_cleaned.query('Quantity > 0') # Only get customers where purchase totals were positive

```

Final resulting matrix of grouped purchases is as follows:

grouped_purchased

	CustomerID	StockCode	Quantity
0	12346	23166	1
1	12347	16008	24
2	12347	17021	36
3	12347	20665	6
4	12347	20719	40
...
267610	18287	72351B	24
267611	18287	84507C	6
267612	18287	85039A	96
267613	18287	85039B	120
267614	18287	85040A	48

266723 rows × 3 columns

Instead of representing an explicit rating, the purchase quantity can represent a “confidence” in terms of how strong the interaction was. Items with a larger number of purchases by a customer can carry more weight in our ratings matrix

of purchases. Our last step is to create the sparse ratings matrix of users and items.

```
customers = list(np.sort(grouped_purchased.CustomerID.unique())) # Get our unique customers
products = list(grouped_purchased.StockCode.unique()) # Get our unique products that were purchased
quantity = list(grouped_purchased.Quantity) # All of our purchases

from pandas.api.types import CategoricalDtype
rows = grouped_purchased.CustomerID.astype( CategoricalDtype(categories = customers)).cat.codes
# Get the associated row indices
cols = grouped_purchased.StockCode.astype(CategoricalDtype ( categories = products)).cat.codes
# Get the associated column indices
purchases_sparse = sparse.csr_matrix((quantity, (rows, cols)), shape=(len(customers), len(products)))
```

Final matrix object:

```
purchases_sparse
```

```
<4338x3664 sparse matrix of type '<class 'numpy.int64'>'
  with 266723 stored elements in Compressed Sparse Row format>
```

We have 4338 customers with 3664 items. For these user/item interactions, 266723 of these items had a purchase.

```
matrix_size = purchases_sparse.shape[0]*purchases_sparse.shape[1] # Number of possible interactions in the matrix
num_purchases = len(purchases_sparse.nonzero()[0]) # Number of items interacted with
sparsity = 100*(1 - (num_purchases/matrix_size))
sparsity
```

```
98.32190920694744
```

98.3% of the interaction matrix is sparse. For collaborative filtering to work, the maximum sparsity we could get away with would probably be about 99.5% or so.

We are well below this, so we should be able to get decent results.

5. Creating a Training and Validation Set

With collaborative filtering, we need all of the user/item interactions to find the proper matrix factorization. We hide a certain percentage of the user/item interactions from the model during the training phase chosen at random. Then, check during the test phase how many of the items that were recommended the user actually ended up purchasing in the end.

Our test set is an exact copy of our original data. The training set, however, will mask a random percentage of user/item interactions and act as if the user never purchased the item (making it a sparse entry with a zero). We then check in the test set which items were recommended to the user that they ended up actually purchasing. If the users frequently ended up purchasing the items most recommended to them by the system, we can conclude the system seems to be working.

As an additional check, we can compare our system to simply recommending the most popular items to every user (beating popularity is a bit difficult). This will be our baseline. This method of testing isn't necessarily the correct answer, because it depends on how we want to use the recommender system.

```

import random
import implicit

def make_train(ratings, pct_test = 0.2):

    test_set = ratings.copy() # Make a copy of the original set to be the test set.
    test_set[test_set != 0] = 1 # Store the test set as a binary preference matrix
    training_set = ratings.copy() # Make a copy of the original data we can alter as our training set.
    nonzero_inds = training_set.nonzero() # Find the indices in the ratings data where an interaction exists
    nonzero_pairs = list(zip(nonzero_inds[0], nonzero_inds[1])) # Zip these pairs together of user,item index into list
    random.seed(0) # Set the random seed to zero for reproducibility
    num_samples = int(np.ceil(pct_test*len(nonzero_pairs))) # Round the number of samples needed to the nearest integer
    samples = random.sample(nonzero_pairs, num_samples) # Sample a random number of user-item pairs without replacement
    user_inds = [index[0] for index in samples] # Get the user row indices
    item_inds = [index[1] for index in samples] # Get the item column indices
    training_set[user_inds, item_inds] = 0 # Assign all of the randomly chosen user-item pairs to zero
    training_set.eliminate_zeros() # Get rid of zeros in sparse array storage after update to save space
    return training_set, test_set, list(set(user_inds)) # Output the unique list of user rows that were altered

product_train, product_test, product_users_altered = make_train(purchases_sparse, pct_test = 0.2)

```

The function 'make_train' will take in the original user-item matrix and "mask" a percentage of the original ratings where a user-item interaction has taken place for use as a test set.

This will return our training set, a test set that has been binarized to 0/1 for purchased/not purchased, and a list of which users had at least one item masked.

We will test the performance of the recommender system on these users only.

We are masking 20% of the user/item interactions. Now that we have our train/test split, it is time to implement the alternating least squares from the paper.

6. Implementing ALS for Implicit Feedback

In the research paper, we can see the key equations we need to implement into the algorithm. First, we have our ratings matrix which is sparse (represented by the `product_train` sparse matrix object). We need to turn this into a confidence matrix (from page 4):

$$c_{ui} = 1 + \alpha r_{ui}$$

Where C_{ui} is the confidence matrix for our users u and our items i . The α term represents a linear scaling of the rating preferences (in our case number of purchases) and the r_{ui} term is our original matrix of purchases. The paper suggests 40 as a good starting point. After taking the derivative of equation 3 in the paper, we can minimize the cost function for our users U :

$$x_u = (Y^T C^u Y + \lambda I)^{-1} Y^T C^u p(u)$$

We can derive a similar equation for our items:

$$y_i = (X^T X + X^T (C^i - I) X + \lambda I)^{-1} X^T C^i p(i)$$

These will be the two equations we will iterate back and forth between until they converge. We also have a regularization term λ that can help prevent overfitting

during the training stage as well, along with our binarized preference matrix p which is just 1 where there was a purchase (or interaction) and zero where there was not. The code for the above mentioned math part is as shown below.

```
def implicit_weighted_ALS(training_set, lambda_val = 0.1, alpha = 40, iterations = 10, rank_size = 20, seed = 0):  
    conf = (alpha*training_set) # To allow the matrix to stay sparse, I will add one later when each row is taken  
                                # and converted to dense.  
    num_user = conf.shape[0]  
    num_item = conf.shape[1] # Get the size of our original ratings matrix, m x n  
  
    # initialize our X/Y feature vectors randomly with a set seed  
    rstate = np.random.RandomState(seed)  
  
    X = sparse.csr_matrix(rstate.normal(size = (num_user, rank_size))) # Random numbers in a m x rank shape  
    Y = sparse.csr_matrix(rstate.normal(size = (num_item, rank_size))) # Normally this would be rank x n but we can  
                                # transpose at the end. Makes calculation more simple.  
  
    X_eye = sparse.eye(num_user)  
    Y_eye = sparse.eye(num_item)  
    lambda_eye = lambda_val * sparse.eye(rank_size) # Our regularization term lambda*I.  
    |
```

```

for iter_step in range(iterations): # Iterate back and forth between solving X given fixed Y and vice versa
    # Compute yTy and xTx at beginning of each iteration to save computing time
    yTy = Y.T.dot(Y)
    xTx = X.T.dot(X)
    # Being iteration to solve for X based on fixed Y
    for u in range(num_user):
        conf_samp = conf[u,:].toarray() # Grab user row from confidence matrix and convert to dense
        pref = conf_samp.copy()
        pref[pref != 0] = 1 # Create binarized preference vector
        CuI = sparse.diags(conf_samp, [0]) # Get Cu - I term, don't need to subtract 1 since we never added it
        yTCuIY = Y.T.dot(CuI).dot(Y) # This is the yT(Cu-I)Y term
        yTCuPu = Y.T.dot(CuI + Y_eye).dot(pref.T) # This is the yTCuPu term, where we add the eye back in
                                                    # Cu - I + I = Cu
        X[u] = spsolve(yTy + yTCuIY + lambda_eye, yTCuPu)
        # Solve for Xu = ((yTy + yT(Cu-I)Y + lambda*I)^-1)yTCuPu, equation 4 from the paper
    # Begin iteration to solve for Y based on fixed X
    for i in range(num_item):
        conf_samp = conf[:,i].T.toarray() # transpose to get it in row format and convert to dense
        pref = conf_samp.copy()
        pref[pref != 0] = 1 # Create binarized preference vector
        CiI = sparse.diags(conf_samp, [0]) # Get Ci - I term, don't need to subtract 1 since we never added it
        xTCiIX = X.T.dot(CiI).dot(X) # This is the xT(Cu-I)X term
        xTCiPi = X.T.dot(CiI + X_eye).dot(pref.T) # This is the xTCiPi term
        Y[i] = spsolve(xTx + xTCiIX + lambda_eye, xTCiPi)
        # Solve for Yi = ((xTx + xT(Cu-I)X) + lambda*I)^-1)xTCiPi, equation 5 from the paper
    # End iterations
return X, Y.T # Transpose at the end to make up for not being transposed at the beginning.
                # Y needs to be rank x n. Keep these as separate matrices for scale reasons.

```

Let's try just a single iteration of the code to see how it works. I will choose 20 latent factors as my rank matrix size along with an alpha of 15 and regularization of 0.1 (which I found in testing does the best).

```

user_vecs, item_vecs = implicit_weighted_ALS(product_train, lambda_val = 0.1, alpha = 15, iterations = 1,
                                              rank_size = 20)

```

```

user_vecs[0,:].dot(item_vecs).toarray()[0,:5]

```

```

array([ 0.01218043, -0.00251753,  0.00943813,  0.00054114,  0.02303351])

```

We can investigate ratings for a particular user by taking the dot product between the user and item vectors (U and V). In the above code, we have it for our first user alone. This is a sample of the first five items out of the 3664 in our stock. The

first user in our matrix has the fifth item with the greatest recommendation out of the first five items. Later, we Speeded Up ALS.

7. Evaluating the Recommender System

20% of the purchases which was masked can be used for testing. Essentially, we need to see if the order of recommendations given for each user matches the items they ended up purchasing. A commonly used metric for this kind of problem is **the area under the Receiver Operating Characteristic (or ROC) curve**.

A greater area under the curve means we are recommending items that end up being purchased near the top of the list of recommended items. In order to do that, we have a function that can calculate a mean area under the curve (AUC) for any user that had at least one masked item. As a benchmark, we will also calculate what the mean AUC would have been if we had simply recommended the most popular items.

```
from sklearn import metrics

def auc_score(predictions, test):
    fpr, tpr, thresholds = metrics.roc_curve(test, predictions)
    return metrics.auc(fpr, tpr)
```

The function below will calculate the AUC for each user in our training set that had at least one item masked. It should also calculate AUC for the most popular items for our users to compare.

```
def calc_mean_auc(training_set, altered_users, predictions, test_set):

    store_auc = [] # An empty list to store the AUC for each user that had an item removed from the training set
    popularity_auc = [] # To store popular AUC scores
    pop_items = np.array(test_set.sum(axis = 0)).reshape(-1) # Get sum of item interactions to find most popular
    item_vecs = predictions[1]
    for user in altered_users: # Iterate through each user that had an item altered
        training_row = training_set[user,:].toarray().reshape(-1) # Get the training set row
        zero_inds = np.where(training_row == 0) # Find where the interaction had not yet occurred
        # Get the predicted values based on our user/item vectors
        user_vec = predictions[0][user,:]
        pred = user_vec.dot(item_vecs).toarray()[0,zero_inds].reshape(-1)
        # Get only the items that were originally zero
        # Select all ratings from the MF prediction for this user that originally had no interaction
        actual = test_set[user,:].toarray()[0,zero_inds].reshape(-1)
        # Select the binarized yes/no interaction pairs from the original full data
        # that align with the same pairs in training
        pop = pop_items[zero_inds] # Get the item popularity for our chosen items
        store_auc.append(auc_score(pred, actual)) # Calculate AUC for the given user and store
        popularity_auc.append(auc_score(pop, actual)) # Calculate AUC using most popular and score
    # End users iteration

    return float('%.3f'%np.mean(store_auc)), float('%.3f'%np.mean(popularity_auc))
# Return the mean AUC rounded to three decimal places for both test and popularity benchmark

We can now use this function

calc_mean_auc(product_train, product_users_altered,
               [sparse.csr_matrix(user_vecs), sparse.csr_matrix(item_vecs.T)], product_test)
# AUC for our recommender system

(0.868, 0.814)
```

Our system had a mean **AUC of 0.87**, while the popular item benchmark had a lower AUC of 0.814.

8. A Recommendation Example (Check)

Let's examine the recommendations given to a particular user and decide subjectively if they make any sense.

```
customers_arr = np.array(customers) # Array of customer IDs from the ratings matrix
products_arr = np.array(products) # Array of product IDs from the ratings matrix
```

```
def get_items_purchased(customer_id, mf_train, customers_list, products_list, item_lookup):
    cust_ind = np.where(customers_list == customer_id)[0][0] # Returns the index row of our customer id
    purchased_ind = mf_train[cust_ind,:].nonzero()[1] # Get column indices of purchased items
    prod_codes = products_list[purchased_ind] # Get the stock codes for our purchased items
    return item_lookup.loc[item_lookup.StockCode.isin(prod_codes)]
```

```
customers_arr[:5]
```

```
array([12346, 12347, 12348, 12349, 12350])
```

```
get_items_purchased(12346, product_train, customers_arr, products_arr, item_lookup)
```

	StockCode	Description
61619	23166	MEDIUM CERAMIC TOP STORAGE JAR

We can see that the first customer listed has an ID of 12346. He/She has purchased a ceramic jar for storage, medium size.

```
def rec_items(customer_id, mf_train, user_vecs, item_vecs, customer_list, item_list, item_lookup, num_items = 10):

    cust_ind = np.where(customer_list == customer_id)[0][0] # Returns the index row of our customer id
    pref_vec = mf_train[cust_ind,:].toarray() # Get the ratings from the training set ratings matrix
    pref_vec = pref_vec.reshape(-1) + 1 # Add 1 to everything, so that items not purchased yet become equal to 1
    pref_vec[pref_vec > 1] = 0 # Make everything already purchased zero
    rec_vector = user_vecs[cust_ind,:].dot(item_vecs.T) # Get dot product of user vector and all item vectors
    # Scale this recommendation vector between 0 and 1
    min_max = MinMaxScaler()
    rec_vector_scaled = min_max.fit_transform(rec_vector.reshape(-1,1))[:,0]
    recommend_vector = pref_vec*rec_vector_scaled
    # Items already purchased have their recommendation multiplied by zero
    product_idx = np.argsort(recommend_vector)[::-1][:num_items] # Sort the indices of the items into order
    # of best recommendations
    rec_list = [] # start empty list to store items
    for index in product_idx:
        code = item_list[index]
        rec_list.append([code, item_lookup.Description.loc[item_lookup.StockCode == code].iloc[0]])
        # Append our descriptions to the list
    codes = [item[0] for item in rec_list]
    descriptions = [item[1] for item in rec_list]
    final_frame = pd.DataFrame({'StockCode': codes, 'Description': descriptions}) # Create a dataframe
    return final_frame[['StockCode', 'Description']] # Switch order of columns around
```

```
rec_items(12346, product_train, user_vecs, item_vecs, customers_arr, products_arr, item_lookup,
          num_items = 10)
```

Essentially, this will retrieve the N highest ranking dot products between our user and item vectors for a particular user. Items already purchased are not recommended to the user. For now, let's use a default of 10 items.

	StockCode	Description
0	23167	SMALL CERAMIC TOP STORAGE JAR
1	23165	LARGE CERAMIC TOP STORAGE JAR
2	22963	JAM JAR WITH GREEN LID
3	22962	JAM JAR WITH PINK LID
4	23294	SET OF 6 SNACK LOAF BAKING CASES
5	23236	DOILEY STORAGE TIN
6	23295	SET OF 12 MINI LOAF BAKING CASES
7	23168	CLASSIC SUGAR DISPENSER
8	23293	SET OF 12 FAIRY CAKE BAKING CASES
9	23296	SET OF 6 TEA TIME BAKING CASES

These recommendations seem quite good! The recommendation system has no real understanding of what a ceramic jar is. All it knows is the purchase history. It identified that people purchasing a medium sized jar may also want to purchase jars of a differing size. The recommender system also suggests jar magnets and a sugar dispenser, which is similar in use to a storage jar.

9. Conclusion

The dataset considered is online retail dataset which comes under the category of implicit feedback datasets. The recommendation system which was built using

alternating least squares method works good which was tested against many examples few of which are shown in the video.

10. References

[1] Yifan Hu, Yehuda Koren, Chris Volinsky ., Collaborative Filtering for Implicit Feedback Datasets.

[2] Mukta kohar, Chhavi Rana Department of computer science and Engg U.I.E.T, MDU, Rohtak., Survey Paper on Recommendation System.